Zellic

Tortuga Liquid Staking

Smart Contract Security Assessment

October 21, 2022

Prepared by:

Aaron Esau and Varun Verma

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Move Labs from September 7th to September 20th, 2022.

Our general overview of the code is that clarity was only slightly impacted because of the organization; the relationships between the modules were complicated. The code coverage is high, and tests are included for the majority of the functions. No prover specifications were written at the time of the audit. The documentation was acceptable but could be improved. The code was relatively easy to comprehend.

We applaud Move Labs for their attention to detail and diligence in maintaining high code quality standards in the development of Tortuga Liquid Staking.

Zellic thoroughly reviewed the Tortuga Liquid Staking codebase to find protocol-breaking bugs as defined by the whitepaper and to find any technical issues outlined in the Methodology section (2.2) of this document.

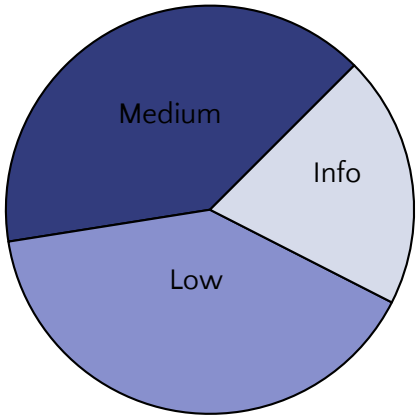
Specifically, taking into account Tortuga Liquid Staking's threat model, we focused heavily on issues caused by rounding or precision errors preventing commissions or payouts from occurring, locking funds, or enabling griefing attacks.

During our assessment on the scoped Tortuga Liquid Staking contracts, we discovered five findings. Fortunately, no critical issues were found. Of the five findings, two were of medium severity, two were of low severity, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for Move Labs's benefit in the Discussion section (5) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	2
Informational	1



2 Introduction

2.1 About Tortuga Liquid Staking

Tortuga is a liquid staking protocol built on top of the Aptos blockchain. It allows Aptos users to stake APT coins to help secure the Aptos chain while earning rewards and maintaining liquidity.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We

look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Tortuga Liquid Staking Contracts

Repository	https://github.com/MoveLabsXYZ/liquid-staking
Versions	`fbdb74f1c6835d1780630c82f301fd573d295427`
Programs	• ./tortuga/sources/math.move • ./tortuga/sources/delegation_service.move . • /tortuga/sources/stake_router.move . • /tortuga/sources/delegation_state.move . • /tortuga/sources/validator_states.move . • /tortuga/sources/stake_pool_helpers.move . • /tortuga/sources/test_helpers.move . • /tortuga/sources/circular_buffer.move . • /tortuga/sources/staked_aptos.move . • /tortuga/sources/validator_router.move
Type	Move
Platform	Aptos

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

Varun Verma, Engineer
varun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 7, 2022	Start of primary review period
September 21, 2022	End of primary review period

3 Detailed Findings

3.1 Tortuga coin initialization

- **Target**: tortuga::initialize_tortuga_liquid_staking
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Medium
- **Impact**: Medium

Description

The `initialize_tortuga_liquid_staking` function calls `coin::initialize` to instantiate the `Coin` resource. However, within the function body of `coin::initialize` is an assertion statement that the creator of the resource matches the deploying package's address.

```
assert!(
    coin_address<CoinType>() == account_addr,
    error::invalid_argument(ECOIN_INFO_ADDRESS_MISMATCH),
);
```

Impact

Users would not be able to access this function and not deploy their own version of `StakedAptosCoin`.

Recommendations

We recommend making this function only accessible for Tortuga's address.

Remediation

Move Labs fixed this issue in commit `ef89a88`.

3.2 Protocol configurations

- **Target**: tortuga::stake_router.move
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Low
- **Impact**: Medium

Description

The following setter functions configure the protocol but have no input validation: `set_min_transaction_amount`, `set_reward_commission`, and `set_cooldown_period`.

```
public entry fun set_reward_commission(
    tortuga: &signer,
    value: u64
) acquires StakingStatus {
    let staking_status =
    borrow_global_mut<StakingStatus>(signer::address_of(tortuga));
    staking_status.reward_commission = value;
}

public entry fun set_cooldown_period(
    tortuga: &signer,
    value: u64
) acquires StakingStatus {
    let staking_status =
    borrow_global_mut<StakingStatus>(signer::address_of(tortuga));
    staking_status.cooldown_period = value;
}

public entry fun set_min_transaction_apt_amount(
    tortuga: &signer,
    value: u64
) acquires StakingStatus {
    let staking_status =
    borrow_global_mut<StakingStatus>(signer::address_of(tortuga));
    staking_status.min_transaction_apt_amount = value;
}
```

Impact

This could pose as a centralization risk and allow impractical configuration values.

For example, setting the minimum transaction amount too high could inhibit new users from entering the protocol, and setting the reward commission too high mistakingly would inhibit validators from being able to acquire reasonable amounts of delegations.

Recommendations

We recommend adding upper bound checks on these functions to allow for a reasonable max threshold.

Remediation

Move Labs fixed this issue in commit `ef89a88`.

3.3 Payouts round down

- **Target**: tortuga::delegation_state
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

It is possible to perform an economically impractical, griefing-style attack that abuses the rounding down behavior of `mul_div` in `disperse_all_payouts` to ensure only those with a relatively high number of shares can receive a payout:

```
let payout_value = math::mul_div(
    delegator_shares_for_payout,
    reserve_balance,
    reserved_share_supply,
);
```

If the `reserve_balance` is low enough, delegators with few shares would receive zero payout while delegators with many shares would receive some. Dust is refunded to the reserve at the end of `disperse_all_payouts`, meaning repeated, quick calls to `disperse_all_payouts` would result in only high-value delegators getting payouts.

Impact

Malicious, high-value delegators (i.e., those with many shares) could cause lower-value delegators to not receive any payouts.

Recommendations

A potential solution could be to delay payout until a minimum reserve balance is met.

Remediation

Move Labs fixed this issue in commit `ef89a88`.

3.4 Centralization risk in minimum delegation amount

- **Target**: delegation::delegation_service
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Medium
- **Impact**: Low

Description

The `set_min_delegation_amount` function allows pool owners to set an arbitrary value for the minimum delegation amount without any constraints. So, a pool owner could set the value to the maximum `u64`, effectively making it impossible for anyone except the owner or protocol to delegate APT to a `managed_stake_pool`.

```
public entry fun set_min_delegation_amount(pool_owner: &signer, value:
    u64) acquires ManagedStakePool {
    let managed_pool_address = signer::address_of(pool_owner);
    let managed_stake_pool =
    borrow_global_mut<ManagedStakePool>(managed_pool_address);
    managed_stake_pool.min_delegation_amount = value;
}
```

Impact

A pool owner could set the value to the maximum `u64`, effectively making it impossible for anyone except the owner or protocol to delegate APT to a `managed_stake_pool`.

Recommendations

Set a hardcoded maximum value for the `min_delegation_amount`.

Remediation

Move Labs fixed this issue in commit `ef89a88`.

3.5 Precision loss in reward rate calculation

- **Target**: oracle::validator_states
- **Category**: Coding Mistakes
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

When calculating the effective reward rate, the `effective_reward_rate` function uses an order of operations that is not ideal; we recommend multiplying before dividing in cases where there is little risk of overflow to improve calculation precision.

Impact

The effective reward rate may be slightly lower than intended.

Recommendations

Change the order of the following operations:

```
fun effective_reward_rate(
    stats_config: &StatsConfig,
    rewards: u128,
    balance_at_last_update: u128,
    time_delta: u128,
): u128 {
    (rewards * stats_config.rate_normalizer / balance_at_last_update) *
        stats_config.time_normalizer / time_delta

    (rewards * stats_config.rate_normalizer*stats_config.time_normalizer)/
        (balance_at_last_update * time_delta)
}
```

Remediation

In response to this finding, Move Labs noted that:

> We have two normalizers just so that we can have double control over precision. rate_normalizer will be as large as possible that still ensures no overflows

> in the first mul_div. Then time_normalizer could be any other reasonable value for precision.
> Multiplying the normalizers first, as in the recommendation is the same as using just one normalizer. We are hoping to get additional precision if necessary using two normalizers.

4 Formal Verification

The MOVE prover allows for formal specifications to be written on MOVE code, which can provide guarantees on function behavior.

During the audit period, we provided Move Labs with Move prover specifications, a form of formal verification. We found the prover to be highly effective at evaluating the entirety of certain functions' behavior and recommend the Move Labs team to add more specifications to their code base.

One of the issues we encountered was that the prover does not support recursive code yet. We suggest replacing the recursive functions, specifically the `math::pow` functions to a loop form so additional specs can be written on the project.

The following is a sample of the specifications provided.

4.1 tortuga::stake_router

Verifies the result is a multiplication-divide:

```
spec calc_shares_to_value {
    requires t_apt_supply ≠ 0;
    aborts_if t_apt_supply < num_shares;
    ensures result ≤ MAX_U64;
    ensures result == num_shares * total_worth / t_apt_supply;
}
```

Verifies the following resources are created upon initialization:

```
spec initialize_tortuga_liquid_staking  {
    ensures exists<StakedAptosCapabilities>(signer::address_of(tortuga));
    ensures exists<StakingStatus>(signer::address_of(tortuga));
    ensures
    exists<validator_router::Status>(signer::address_of(tortuga));
    ensures
    exists<validator_router::DelegationAccounts>(signer::address_of(tortuga));
}
```

Verifies values were mutated:

```
spec set_min_transaction_amount {
    ensures
    borrow_global_mut<StakingStatus>(signer::address_of(tortuga)).min_transaction_amount
    == value;
}

spec set_cooldown_period {
    ensures
    borrow_global_mut<StakingStatus>(signer::address_of(tortuga)).cooldown_period
    == value;
}

spec set_reward_commission {
    ensures
    borrow_global_mut<StakingStatus>(signer::address_of(tortuga)).reward_commission
    == value;
}
```

4.2 helpers::circular_buffer

Verifies the buffer always contains the latest value pushed:

```
spec push {
    ensures len(old(cbuffer.buffer)) < max_length && cbuffer.last_index +
    1 > len(cbuffer.buffer) ⟹ contains(cbuffer.buffer, value);
}
```

Verifies the empty function returns an empty buffer:

```
spec empty {
    ensures len(result.buffer) == 0;
    ensures result.last_index == 0;
}
```

Verifies the length of `cbuffer`:

```
spec length {
```

```
    ensures len(cbuffer.buffer) == result;
}
```

Verifies `borrow_oldest` and `round_robin` behavior:

```
spec fun helper_round_robin(a: u64, b: u64): u64 {
    assert!(b > 0 && a ≤ b, error::invalid_argument(EARITHMETIC_ERROR));
    if (a < b) {
        a
    }
    else {
        0
    }
}
spec round_robin {
    aborts_if b > 0 || a ≤ b;
}

spec borrow_oldest {
    // Verifies behavior about the borrow_oldest function in
    circular_buffer
    aborts_if cbuffer.last_index + 1 > len(cbuffer.buffer);
    aborts_if len(cbuffer.buffer) == 0;
    let oldest_index = helper_round_robin(cbuffer.last_index +1,
    len(cbuffer.buffer));
    ensures result == cbuffer.buffer[oldest_index];
}
```

4.3 tortuga::stakedaptoscoin

Verifies `StakedAptosCoin` exists after initialization:

```
spec register_for_t_apt {
    ensures
    exists<coin::CoinStore<StakedAptosCoin>>(signer::address_of(account));
}
```

4.4 helpers::math

Verifies when `mul_div` aborts and the resulting output:

```
spec mul_div {
    aborts_if c == 0;
    aborts_if a * b / c > MAX_U64;
    ensures result ≤ MAX_U64;
}
```

Verifies it never aborts, thus actually safe:

```
spec safe_sub_u128 {
    aborts_if false;
}
```

5 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

5.1 Evolving nature of Aptos core

While the Aptos blockchain prepares for its upcoming mainnet launch in autumn, periodically breaking changes are introduced to `aptos-stdlib` and `aptos-framework`. We suspect this will continue to occur, even shortly after launch. We recommend the Tortuga team to stay up to date with any changes that may occur, with a specific attention to the `stake.move` file in `aptos-framework`, which governs validator behavior.

5.2 Griefing

Certain aspects of the protocol iterate over data structures, for example in `tortuga::delegation_state`,

```
while (pool::num_share_holders(&shares_data.reserved_pool) > 0)
```

which pose as a danger for gas limit errors. In this particular instance, Tortuga mitigated the concern by providing an upperbound of 100 on the number of shareholders. Nevertheless, gas metrics on Aptos are still relatively unclear, and an amount of shareholders close to the maximum limit could pose a threat of out of gas errors.

5.3 Simple map griefing

The Simple Map data structure is susceptible to out of gas concerns, which potentially could cause an issue if `unclaimed_stake_pool_owner_caps` gets too large.

Adding a time constraint for an individual to claim their owner cap could mitigate this risk.

5.4 Integration and composability

To improve the interoperability of the protocol within the Aptos ecosystem, various methods for accessing resources from other contracts may be beneficial. For exam-

ple, a getter method on the number of tickets a delegator has could be useful.

One area in which the protocol achieved composability was via the following function:

```
public fun stake_coins(
        coins_to_stake: coin::Coin<AptosCoin>
    ): coin::Coin<StakedAptosCoin>
```

And we believe adding more *secure* integration pathways could be beneficial to the success of the protocol.

5.5 Resource inconsistency

Within the code are resources that can be acquired by normal users that should be ideally only be reserved for admin acquisition only. This does not pose as an immediate security risk, however the getter methods for these resources would not work.

For instance, one could acquire the `StakingStatus` resource that exists on the publically available function `initialize_tortuga_liquid_staking`.

However this following function, which utilizes a getter for this resource

```
public fun get_total_worth(): u64 acquires StakingStatus {
    let staking_status = borrow_global<StakingStatus>(@tortuga);
    let unclaimed_balance =
        staking_status.total_claims_balance -
            staking_status.total_claims_balance_cleared;
    validator_router::get_total_balance() - (unclaimed_balance as u64)
}
```

acquires the resource strictly from the address of `@tortuga`, rendering the ability for a user to have their own StakingStatus resource impractical.

We suggest the initialization functions to be accessible only for the address of `@tortuga`.

6 Audit Results

At the time of our audit, the code was not deployed to Aptos Mainnet as the blockchain had not been launched yet.

During our audit, we discovered five findings. Of these, two were medium risk, two were low risk and one was a suggestion (informational). Move Labs acknowledged all findings and implemented fixes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.